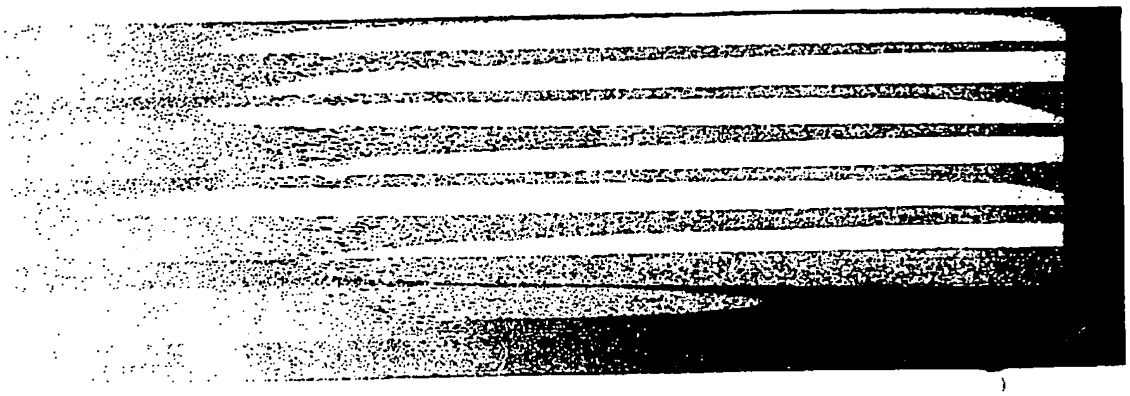
Exhibit A



*Java™ Examples in a Nutshell, Second Edition*
by David Flanagan

Published by O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472.

**Editor:** Paula Ferguson

**Production Editor:** Mary Anne Weeks Mayo

**Cover Designer:** Edie Freedman

**Printing History:**

September 1997: First Edition.

September 2000: Second Edition.

*Library of Congress Cataloging-in-Publication Data*

Flanagan, David.
Java examples in a nutshell / David Flanagan.--2nd ed.
p.cm.-- (A Nutshell handbook) (The Java series)
Includes index.
ISBN 1-596-00039-1
1.Java (Computer program language) I.Title. II.Series. III.Java series (O'Reilly & Associates)
QA76.73.J38 F5518 2000
005.13'3--dc21 00-045274

ISBN: 0-596-00039-1
[M]

## A Rectangle Class

Example 2-1 shows a class that represents a rectangle. Each instance of this Rect class has four fields, x1, y1, x2, and y2, that define the coordinates of the corners of the rectangle. The Rect class also defines a number of methods that operate on those coordinates.

Note the toString() method. This method overrides the toString() method of java.lang.Object, which is the implicit superclass of the Rect class. toString() produces a String that represents a Rect object. As you'll see, this method is quite useful for printing out Rect values.

*Example 2-1: Rect.java*

```
package com.davidflanagan.examples.classes;
/**
 * This class represents a rectangle.  Its fields represent the coordinates
 * of the corners of the rectangle.  Its methods define operations that can
 * be performed on Rect objects.
 **/
public class Rect {
    // These are the data fields of the class
    public int x1, y1, x2, y2;

    /**
     * The is the main constructor for the class.  It simply uses its arguments
     * to initialize each of the fields of the new object.  Note that it has
     * the same name as the class, and that it has no return value declared in
     * its signature.
     **/
    public Rect(int x1, int y1, int x2, int y2) {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
    }

    /**
     * This is another constructor.  It defines itself in terms of the above
     **/
    public Rect(int width, int height) { this(0, 0, width, height); }

    /** This is yet another constructor. */
    public Rect() { this(0, 0, 0, 0); }

    /** Move the rectangle by the specified amounts */
    public void move(int deltax, int deltay) {
        x1 += deltax; x2 += deltax;
        y1 += deltay; y2 += deltay;
    }

    /** Test whether the specified point is inside the rectangle */
    public boolean isInside(int x, int y) {
        return ((x >= x1) && (x <= x2) && (y >= y1) && (y <= y2));
    }

    /**
     * Return the union of this rectangle with another.  I.e. return the
     * smallest rectangle that includes them both.
```

*Example 2–1: Rect.java (continued)*

```
    **/
    public Rect union(Rect r) {
        return new Rect((this.x1 < r.x1) ? this.x1 : r.x1,
                        (this.y1 < r.y1) ? this.y1 : r.y1,
                        (this.x2 > r.x2) ? this.x2 : r.x2,
                        (this.y2 > r.y2) ? this.y2 : r.y2);
    }

    /**
     * Return the intersection of this rectangle with another.
     * I.e. return their overlap.
     **/
    public Rect intersection(Rect r) {
        Rect result =  new Rect((this.x1 > r.x1) ? this.x1 : r.x1,
                                (this.y1 > r.y1) ? this.y1 : r.y1,
                                (this.x2 < r.x2) ? this.x2 : r.x2,
                                (this.y2 < r.y2) ? this.y2 : r.y2);
        if (result.x1 > result.x2) { result.x1 = result.x2 = 0; }
        if (result.y1 > result.y2) { result.y1 = result.y2 = 0; }
        return result;
    }

    /**
     * This is a method of our superclass, Object.  We override it so that
     * Rect objects can be meaningfully converted to strings, can be
     * concatenated to strings with the + operator, and can be passed to
     * methods like System.out.println()
     **/
    public String toString() {
        return "[" + x1 + "," + y1 + "; " + x2 + "," + y2 + "]";
    }
}
```

## Testing the Rect Class

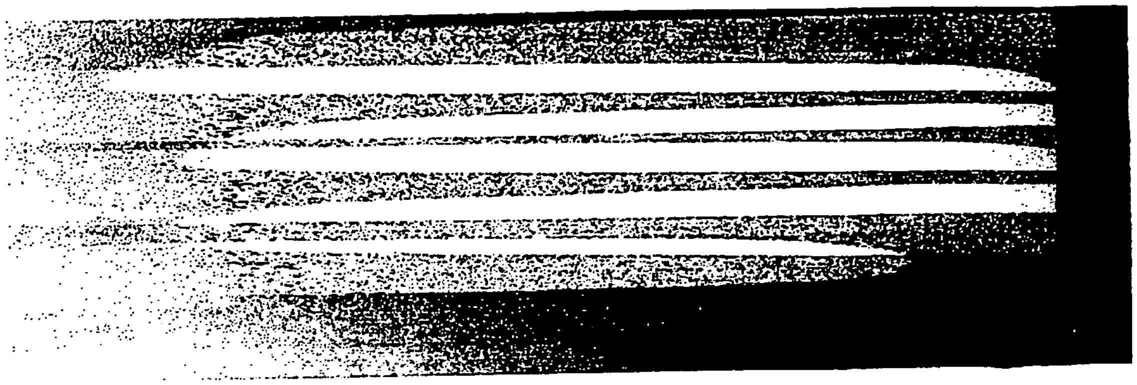
Example 2-2 is a standalone program named RectTest that puts the Rect class of Example 2-1 through its paces. Note the use of the new keyword and the Rect() constructor to create new Rect objects. The program uses the . operator to invoke methods of the Rect objects and to access their fields. The test program also relies implicitly on the toString() method of Rect when it uses the string concatenation operator (+) to create strings to be displayed to the user.

*Example 2–2: RectTest.java*

```
package com.davidflanagan.examples.classes;

/** This class demonstrates how you might use the Rect class */
public class RectTest {
    public static void main(String[] args) {
        Rect r1 = new Rect(1, 1, 4, 4);    // Create Rect objects
        Rect r2 = new Rect(2, 3, 5, 6);
        Rect u = r1.union(r2);             // Invoke Rect methods
        Rect i = r2.intersection(r1);

        if (u.isInside(r2.x1, r2.y1))   // Use Rect fields and invoke a method
            System.out.println("(" + r2.x1 + "," + r2.y1 +
```



*Example 2-4: ColoredRect.java*

```
package com.davidflanagan.examples.classes;
import java.awt.*;

/**
 * This class subclasses DrawableRect and adds colors to the rectangle it draws
 **/
public class ColoredRect extends DrawableRect {
    // These are new fields defined by this class.
    // x1, y1, x2, and y2 are inherited from our super-superclass, Rect.
    protected Color border, fill;

    /**
     * This constructor uses super() to invoke the superclass constructor, and
     * also does some initialization of its own.
     **/
    public ColoredRect(int x1, int y1, int x2, int y2,
                       Color border, Color fill)
    {
        super(x1, y1, x2, y2);
        this.border = border;
        this.fill = fill;
    }

    /**
     * This method overrides the draw() method of our superclass so that it
     * can make use of the colors that have been specified.
     **/
    public void draw(Graphics g) {
        g.setColor(fill);
        g.fillRect(x1, y1, (x2-x1), (y2-y1));
        g.setColor(border);
        g.drawRect(x1, y1, (x2-x1), (y2-y1));
    }
}
```

## Complex Numbers

Example 2-5 shows the definition of a class that represents complex numbers. You may recall from algebra class that a complex number is the sum of a real number and an imaginary number. The imaginary number *i* is the square root of –1. This `ComplexNumber` class defines two `double` fields, which represent the real and imaginary parts of the number. These fields are declared `private`, which means they can be used only within the body of the class; they are inaccessible outside the class. Because the fields are inaccessible, the class defines two accessor methods, `real()` and `imaginary()`, that simply return their values. This technique of making fields `private` and defining accessor methods is called *encapsulation*. Encapsulation hides the implementation of a class from its users, which means that you can change the implementation without it affecting the users.

Notice that the `ComplexNumber` class doesn't define any methods, other than the constructor, that set the values of its fields. Once a `ComplexNumber` object is created, the number it represents can never be changed. This property is known as *immutability*; it is sometimes useful to design objects that are immutable like this.